AFRL-IF-RS-TR-2003-305
Final Technical Report
December 2003

ACME: A BASIS FOR ARCHITECTURE EXCHANGE

Teknowledge Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D929

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

APPROVED: /s/
NANCY A. ROBERTS
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** DECEMBER 2003	**3. REPORT TYPE AND DATES COVERED** **FINAL** **Dec 96 – Feb 03**

4. TITLE AND SUBTITLE

ACME: A BASIS FOR ARCHITECTURE EXCHANGE

5. FUNDING NUMBERS

G - F30602-96-2-0224
PE - 62702E
PR - D929
TA - 01
WU - 01

6. AUTHOR(S)

David S. Wile
David Garlan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Teknowledge Corporation
1800 Embarcadero Road
Palo Alto CA 94303

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2003-305

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Nancy Roberts/IFTB/(315) 330-3566 Nancy.Roberts@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

The Acme project began with the goal of providing a common language that could be used to support the interchange of architectural descriptions between a variety of design tools. It remains useful in that role, but since the project's inception the Acme language and its support toolkit have grown into a solid foundation upon which new software architecture design and analysis tools can be built without the need to rebuild standard infrastructure. The Acme Language and the Acme Tool Developer's Library (AcmeLib) provide a generic, extensible infrastructure for describing, representing, generating, and analyzing software architecture descriptions. They provide three fundamental capabilities: (1) A generic interchange format for architectural designs, allowing architectural tool developers to readily integrate their tools with other complementary tools; (2) An extensible foundation and infrastructure, allowing tool builders to avoid needlessly rebuilding standard tooling infrastructure for describing, storing, and manipulating architectural designs; and (3) A useful architecture description language in its own right, providing a straightforward set of language constructs for describing architectural structure, architectural types and styles, and annotated properties of the architectural elements.

14. SUBJECT TERMS Software Architecture, Architecture Description Language, Architecture Definition Language, Architecture Style, Architecture Dynamism, XML, Acme, Formal Specification Language, Architecture Semantics

15. NUMBER OF PAGES 26

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

.

List of Figures

.

Technical Summary:

The use of architectural representations and analyses is a significant enabling technology in support of the evolution of complex systems. The higher level of abstraction provided by architectural description can facilitate understanding of system changes, expose changes that will violate key system-level invariants, and provide a context within which new components can be inserted into an existing system. The Acme project's charter was to develop technology with which software architects can interact to design, view, analyze, debug, and evolve formal architecture specifications. The key to our approach was the design of an architecture interchange language called Acme, translators from domain-specific architecture languages into and out of Acme, and an architecture community-accessible brokerage of storage, browsing, analysis, and simulation tools for architectural descriptions.

Experience within the Domain-Specific Software Architectures community established that no single perspective is appropriate for all architecting activities; each community designed architecting languages with aspects of the domain implicit in the language, thereby facilitating communication with the application engineers in the various domains. On the other hand, certain needs are shared by all architecture design languages (ADLs), such as the need to evolve architectures and their associated artifacts, the need to view the architectures from orthogonal perspectives, and the need to analyze and debug control and data flows. Considerable reinvention of underlying technology was necessary when the Acme project began, because of a lack of consensus in the community -- consensus on graphic interfaces, event languages, simulation primitives, and analysis activities -- none of which is particularly "domain-specific." Moreover, as systems scale, the need to work with mixed ADLs always arises.

There was therefore a critical need to find ways to enable different ADLs and ADL support tools to work together. Acme was originally proposed as an ADL baseline and was to be provided along with a transparently accessible architecture exchange mechanism upon which domain-neutral modeling activities could be built. In the end Acme made it possible to exchange architectural information between a diverse set of architectural development and analysis tools as well as provide a starting point for developing new ADLs. In addition, Acme provides Web access to architectural descriptions and baseline tools for manipulation, analysis, and change-impact analysis of architectural structures that can be universally and transparently invoked from existing ADL platforms. Finally, Acme served as a vehicle for focusing community consensus about architectural representation that evolved over time to accommodate new understandings about requirements for architectural modeling and change.

Among the more important items produced in the project's history were: a complete Acme language specification, with rationale for its inclusions and exclusions; several Acme tools, provided to the Architecture and Generation EDCS Cluster (among these were an Acme description repository, various analyzers for connectedness and completeness, and a translator from Acme into a predicate calculus-based semantics known as Armani); tools that allow others to provide domain-specific analyzers, such as a code-walker and an Acme elaborator -- a tool that translates extended, style-based Acme descriptions into the kernel language; and several web-compliant tools, such as translators into and out of an XML representation for ADLs called xAcme.

In the following, the technical history of the project will be sketched, followed by some of the technology transition success stories and a set of papers inspired by or funded directly during the Acme design and tool development process.

Formative Phase

Acme is a formal specification language for describing software architectures, functioning primarily as an interchange mechanism for translating between other architecture description languages developed by the EDCS community and elsewhere. It was designed to facilitate interoperation of tools using different formal architecture representations. At the same time, its simplicity makes it a neutral publication ADL, in that it can be used to simply represent attributed, typed, bi-partite graph structures. Specifically, Acme

.

specifications comprise a set of *components* with named connection points called *ports*, and *connectors* with connection points called *roles*. A *system* is a collection of components and connectors, together with a specification of how specific roles and ports are *attached* to one another. All of the architectural *elements* – systems, components, connectors, ports, and roles – may be given specific *properties*. These are used to convey any non-topological properties of the architectures described. Acme has additional facilities for strong typing, refinement, and style definition and enforcement; some of these will be discussed extensively in what follows. (See the Appendices A & B for a more thorough Acme overview.)

Early activity involved simply stabilizing the language: insuring that syntactic conventions were applied uniformly, removing redundant syntactic constants, insuring that kernel constructs were truly foundational, etc.

Throughout the project's history considerable attention was paid to accommodating the interests of community stake-holders, insuring that features are present to support the various languages for which Acme was to be used as an interchange medium. The most dramatic change to the language arising from these concerns was the addition of a type system for classifying the primitive architectural entities: components, connectors, ports, and roles. The need for types was certainly made clear by the EDCS Architectural Cluster members in the past, but their introduction was really a result of convincing arguments put forth by Prof. Peter Pepper, who visited ISI from the Technishe Universität in Berlin early in the project's history. The addition of types allowed Acme specifications to rely far less on the very powerful "template" (macro) system of Acme, which was later dropped from active support.

A third area of development was in the specification of "architecture families", dynamic architectures and refinement hierarchies. Early efforts involved experimentation with graph grammars to specify such families. At the same time, David Garlan experimented with a refinement relation notation to reason about components implemented in a style different from their specification.

Early tools included the development and beta release of AcmeLib and the development of a connectivity analysis tool for Acme specifications based on a pipe-and-filter style.

The AcmeLib facility developed at CMU is a support mechanism to allow tool designers to build C++ and Java tools on an abstract representation provided by the library mechanism. Client programs call different methods to create basic elements, such as components and connectors, so the library is actually keeping a dynamic representation of an architecture. This tool suite was extended and generalized throughout the history of the project, later accommodating an XML version of Acme.

Two specific tools were designed to interact through the AcmeLib facility: an Acme-based architecture performance analyzer and a graph layout tool based on (externally developed) state-of-the-art graph layout algorithms.

Considerable concern for "the semantics" of an architecture dominated the project's history. In some sense, a great deal of the appeal of the use of ADLs is that there is very little semantic constraint imposed by an ADL, a priori. Instead, constraints are imposed by defining architecture styles (or families) in which various types are allowed to be used and constraints are put on how they can be connected with one another. Hence, an early achievement was the development of a pipe-and-filter analyzer at ISI, based on a Common Lisp representation of Acme semantics, largely consistent with the AcmeLib facility's abstract representation. The former provides a (virtual) data-base view of an Acme representation while the latter provides a more object-oriented view. These two views were later to be coalesced in the Acme extension called Armani.

More specific details regarding these early technical accomplishments included:

- The release of Acme version 3.0: (1) Cleaner, more uniform syntax; (2) a simple, predicate-based type system for use with both architectural entities (components, connectors, etc) and their property annotations; and (3) support for specification of families of common architectural designs.
- A semantic basis for Acme topological information, type information, and annotated properties was designed and implemented. The semantic interpretation of Acme is based on the relational algebra. In

.

addition, inference rules for Acme types were developed in a separate tool available in the Acme tool suite.

- An initial version of the Acme Tool Developer's Library (Acme-Lib) was made available. Acme-Lib is an object-oriented framework for writing architectural tools that read, write, and manipulate software architecture designs specified in Acme. It supports the rapid development of two classes of applications: (1) tools that translate between "native" ADL's (such as Rapide, Wright, SADL, UniCon, and Aesop) and (2) tools that work directly on Acme descriptions. Included are functions for parsing, unparsing, and manipulating intermediate representations. Tool developers can easily extend Acme-Lib's object-oriented framework to provide functionality that is not provided by the generic Acme-Lib.
- Acme-based tools for Web-based system visualization of Acme system descriptions including automatic graph layout and rudimentary animation.

Early on, the following project *goals, milestones, and evaluation criteria* were developed. We will return at the end of the presentation to evaluate how successfully they were achieved.

- Provide a low buy-in architecture description language for new ADL users.
  Milestones: active use by EDCS community, active use by other DARPA community members, active use by industry.
  Evaluation: problem coverage, relevant analyses provided, ease of use
- Provide network accessible tools for architecture design, analysis (performance prediction, constraint satisfaction, dynamic monitoring, animation, real-time, nondeterministic event), and archiving:
  Milestones: Acme Lib used by outsiders
  Evaluation: perceived leverage over designing the tools from scratch, amount of reuse
- Provide an interchange medium to enable inter-ADL tool use.
  Milestones: active use by EDCS community, active use by other DARPA community members, active use by industry.
  Evaluation: problem coverage, relevant analyses provided, ease of use
- Provide a base level ADL to be extended to new domains.
  Milestones: outside use
  Evaluation: extent of leverage provided by topology, style and dynamic constructs of Acme to such endeavors
- Provide a focus for architecture community consensus.
  Milestones: existing ADLs reformulated as extensions to Acme
  Evaluation: extent of cross-fertilization and community reuse

Maturing Functionality and Support for Architecture Styles

By the end of the second year of the project, two major approaches to providing Acme support had emerged. One, from CMU, supported C++ and Java; another, from ISI, was based on PowerPoint as the user interface and Common Lisp for semantic analysis. The former was usable on unix systems while the latter supported Acme development on the PC. Hence, the Acme "tool suite" comprises a Java/World Wide Web-based portion and a Common Lisp / PowerPoint portion.

The AcmeLib facility was developed at CMU as a support mechanism to allow tool designers to build C++ tools on an abstract representation provided by the library mechanism. Client programs call different methods to create basic elements, such as components and connectors, so the library is actually keeping a dynamic representation of the architecture. An Acme Semantics tool was developed that derives an AcmeLib-like representation from Acme specifications.

The initial ISI contribution was further development of a connectivity analysis tool for Acme specifications based on the Acme Semantics tool created earlier. The novelty introduced here was to base the analysis entirely on the Acme semantics, rather than using the rather involuted mechanism used before. Previously graphics were transformed into syntactic Acme specifications and those into a semantic representation.

.

After this development the graphics could be translated directly into the semantic representation directly, analogous to the AcmeLib representation in unix. This required some extensions to the semantics to deal with architecture representations.

A version of the Acme Manual was developed. It was called the *Acme StrawManual* to emphasize that it contained some proposals that were preliminary in nature: (1) for an expanded constraint language and (2) for extensions to Acme to allow the specification of dynamic architectural aspects. A Common Lisp tool to generate the predicate logic meaning of a dynamic Acme specification had been produced just before these proposals were introduced.

Architecture styles or "families," as they are called in Acme, received a lot of attention during this phase of the project, for a style has considerably greater semantic power than basic Acme. In particular, by limiting the designers' components and connectors to particular types associated with the style, closure-based properties can be reasoned about a priori. For example, by insisting that a pipe and filter style be used exclusively, one can guarantee that there will be no loops allowed in the architectures described. This leverage was used very effectively in the Common Lisp / PowerPoint tool.

Common Lisp / PowerPoint:

The PowerPoint Acme interface started as an experimental version of an interface to an AcmeLib-like facility that represents the topological structure of an architecture using a relational database metaphor. This interface was developed earlier to allow architecture designers to customize tools to particular architecture styles, whose description was either ad hoc or formally specified in Acme. In either case, provisions were provided to allow the interface designer to establish the graphical interface conventions used by the style. Moreover, the different properties one can attach to the various architectural entities – components, connectors, ports, etc. – can be specified through the interface. A mechanism for introducing enumerated types is also provided. This is then interfaced to the Common Lisp analysis suite designed by the architecture designer, who may access the described properties through the relational database representing the architecture instances.

The PowerPoint style development tool was used to specify a *dynamic* architecture style, representing optional and multiple components and connectors. This was in support of experimentation with the proposed set of facilities in the Acme StrawManual. A set of control buttons was added to this style. By putting the buttons in different states, the user can have an analysis made that demonstrates graphically which components and connectors are present when the system is in that state. We also implemented a mechanism to allow analyzers to push information back into the PowerPoint diagrams that are being analyzed, and especially, to modify the attributes on the components.

There was considerable effort to interface well with the community's support for ADL tools. The Acme PowerPoint editor was coordinated via tools using the DCOM protocol rather than an idiosyncratic event language, in preparation for marrying the PowerPoint tools with the Java-based tools via a common event bus. Discussions were begun with the Rapide group on the appropriate set of primitive events for architecture modification, analogous to AcmeLib invocations, and for analyzer result reporting and animation.

Summary of Accomplishments by mid 1998

Designed a layer of dynamic architecture facilities along with minimal extensions to the kernel to allow replicated components and open architecture specifications, characterizing the variety of ways in which the topology can change dynamically.

Developed performance analyzer based on style-specific properties attached to connectors and components (queuing theory in an architecture package).

Designed and implemented an Acme Library facility for use with Common Lisp-based analysis tools (with Synquiry Technologies).

Developed a dynamic architecture semantics generator.

.

Developed an initial version of a PowerPoint style specification mechanism for use in automatic analysis suites designed for problem-domain-specific architecture styles.

Developed Acme PowerPoint editor translating into Acme semantics, with error reports expressed in an analyzer response language.

Developed the Armani constraint language and checker to describe architecture evolution constraints, incorporating dynamic architecture specification facilities in the released Acme and AcmeLib facilities.

Architecture ToolKit

In the fall of 1998 it was decided to take advantage of the considerable consensus of the EDCS architecture community and work toward producing a toolkit that could be used for interoperation among the various community participants, and externally as well.

The Acme project began to develop the toolkit for use by the DARPA community on two broadly-based platforms: the World Wide Web and the Windows NT operating system. It worked toward a capability-based interface to this toolkit to guide users to making the appropriate choices for notations and tools that can support the capabilities they desire, but detailed problems with tool interface and platform assumptions prevented this from ever coming to fruition.

On the NT platform, ISI's PowerPoint-based interface to Acme-topology-based analyzers was used as an interface to notify the analyzers of changes to the topology and as a vehicle for relating analyses back to the architecture diagram. This was built on an idiosyncratic underlying event notification protocol used to communicate architectural and analysis events. This event language was similar in spirit, but different from, the analogous facilities in the AcmeLib implementation at CMU. In support of the toolkit, the Acme project lead community efforts to modify and extend it to become a common API for both NT-based analyzers and Web-based analyzers, so tool developers need only write one version for both platforms, completely independent of the GUIs used to access them and report their analyses. This API design effort was a consensus effort between ISI, CMU, and Stanford University.

After its design, CMU rewrapped the existing analyzers for Acme (and Wright) under separately-provided funding for the toolkit. Stanford began to decompose and wrap the Rapide tool suite to conform to the protocol, under funding provided to them as a subcontractor to ISI and (separately) to CMU.

At this point, the attention began to shift from providing community support for ADL interoperability to providing support to allow usage of Acme as a base-level architecture description language, from which to build domain-specific architecture analyzers, simulators, interpreters and monitors. This emphasis was based on DARPA community (rather than architecture community) involvement; Acme became an entry-level ADL that takes little up-front investment in time to learn and use.[1]

The Acme project took on additional responsibilities to facilitate the creation of a community-wide Architecture ToolKit. The initial toolkit is a loose assembly of references, documentation and web pointers to a variety of tools available from the EDCS community. This assembly constitutes a first-point-of-contact with architecture tools for novice architecture developers. It is available at: http://www.cs.cmu.edu/~spok/adl/.

In support of the Architecture Toolkit, the collection of architectural tools available through the EDCS community was collected, classified, and cataloged. We also designed and prototyped a new Architecture Toolkit integration architecture based on simple message passing connectors. These connectors can be specialized to support both batch and interactive tools, as well as event-based tool integration.

[1] We had also intended to translate Acme into UML styles as the OMG group extended the UML facilities to accommodate architectural connectors and events, but did not complete this translation, since the UML facilities were decided after project support ended.

.

The Armani constraint language and constraint enforcement mechanism were incorporated to constrain the evolutionary development of Acme specifications. The language extensions allow an architect to annotate an architectural design or an Acme family specification with predicates that specify either (a) *invariants*, which must be satisfied under all possible evolutions, or (b) *heuristics*, which are advisory design rules. The tool for checking satisfaction of invariants and heuristics efficiently detects violation of invariants and heuristics.

The design of dynamic architecture specification facilities were completed in an architecture meta-language called AML. With this, architecture designers can characterize and limit how an architecture will change during the execution of the application architecture. The approach is based on specifying the "maximal architectural shell" that covers all expected dynamic variants of an architecture. We extended our PowerPoint semantic editor in use as an architectural editor to provide a "dynamic style" for experimentation with the AML language cited above and for use within the DARPA-funded Embryos project. An analyzer was developed that takes into consideration the state of a system and reflects the instantaneous architecture back onto the diagram of the maximal architectural shell.

The PowerPoint Design Editor was enhanced with a facility for providing animations based on simulations or actual executions of the architecture viewed. The connection with the architecture is through explicit calls on high level primitives such as "change color of object to red" or "move token across connector in 4 seconds."

In support of the core organization of the toolkit, a strawman event language, combining Rapide- and Wright-style events, was designed, but it was not adopted as a de facto standard as hoped. A proposal for its incorporation into Acme itself was never carried through, however.

The community discussed a variety of proposals for the appropriate organizing structure for the Architecture Toolkit. In addition to the documentary website, it was decided to adopt three different organizations for the toolkit, each successive one covering a wider variety of tools, less thoroughly. The most formal connections involve the design of an API for rapid communication among a core set of tools developed at ISI, CMU and Stanford. These integrate parts of the PowerPoint Design Editor (as used on Acme representations), the Acme Tool Suite, and the Rapide event system. A somewhat less formal, but useful classification scheme for a wider set of tools allows tools to communicate via XML. Much less formal still is a document that aids others in the community to assemble a set of tools for their own purposes. The intention of this document is to make precise the APIs required and provided by individual tools in the community, the other tools relied upon, and any platform dependencies of the tools.

Plans to consolidate the community-based Architecture Toolkit around Acme were presented at the DASADA program kickoff. Subsequent support for the development of the Toolkit was through the DASADA program itself, primarily in support for CMU's continuing efforts at organizing and codifying the existing tool suites.

In Support of DASADA

After the Acme subcontract with CMU under David Garlan terminated, subsequent Acme funds were expended in providing architectural support for Teknowledge's role in the DASADA program.

Most of the effort was spent on low-level technical support for the PowerPoint Design Editor facilities needed to provide Acme-based architectural support for various DASADA demonstration projects. For example, we were able to coordinate the Acme PowerPoint editor tools using the DCOM protocol with Java-based analyzers. (Before that our DCOM-based analyzers had been written in C++ and Visual Basic.) Another effort involved redesigning the interface between PowerPoint and the analyzer tools to make them consistent with DCOM standards. Such redesigns entailed re-implementation of the analyzers themselves, also supported by Acme funds.

Simple additions to make the interfaces to the Design Editor more user-friendly were also implemented. For example, the style design interface can provide automatic layout of legends for the styles (detailing the

.

correspondence between component and connector class graphics and names). A simple mechanism to display all of the attributes of a group of objects in a single window was designed and implemented as well, so that all of the textual aspects of the objects can be seen at once. (Previously, special analyzers had to be designed to show the attributes on the diagram or the user had to open and close each object's attribute window to get the same effect, tediously.)

Analyzers for designs expressed using specific architectural styles can be written in any language capable of interfacing with DCOM. Other style-specific tools, like simulators, interpreters, or animators, can be written this way as well. Unfortunately, these tasks require rather technical knowledge of how to program with these interfaces; i.e. the designer must be an expert in Visual Basic, C++, Java, or Haskell. This observation lead to the design and implementation of a generic tool for constraint matching; graphical input is used to represent patterns to be found in presentations and the tool compiles them into code that determines if the patterns match.

In support of a specific DASADA demonstration, we added mechanisms allowing running programs to automatically create architecture specifications in PowerPoint. In addition, an interface between our simulation package and analyzers was designed and implemented to allow statechart simulations as analyzers for PowerPoint Designs. These were used to support creation of an "architectural gauge" that allows one to visualize the dynamic state of a running system's architecture. A facility permitting reanalysis during such dynamic creations was also designed and implemented.

An extension to existing Acme, called Dynamic Acme, was designed to be used to specify architectures whose structure changes during the course of execution of the system. This constitutes a more substantive contribution to the Acme research body. Our DASADA demo used an implementation of this specification in our PowerPoint Design Editor, by using a specific architecture "style." This style extends Acme's normal component-connector vocabulary to include predicates to describe how component and connector copies or versions can evolve, to what extent further refinement and connections can be made, and how and when components and connectors are "identified" with running components within dynamic systems. (This facility is related to Acme Studio's incorporation of constraint specification and maintenance in the Armani extensions.)

We presented a demonstration of how a dynamic architectural style could be used to represent the dynamic process structure of a running system as a DASADA demonstration in Baltimore. This "architectural gauge" was used to demonstrate our "Safe Email" product that prevents inadvertent or malicious access to system resources such as the registry, startup directory, and mail name directory. The process structure is displayed using our PowerPoint Design Editor. The various alerts to malicious access are displayed using indicative colors – e.g. "red" for "danger" – on the boxes representing the processes. We converted that demo into an event-based demo using the community standard Siena event broadcasting system. This allows us to use a separate machine for monitoring email process creation and deletion, along with their access status. We were later able to extend this to monitor several email processes simultaneously, with the potential to detect e.g. systemic attacks, by migrating from the community standard Siena event broadcasting system onto a secure, SSL-based point-to-point communication system.

As Acme funding dwindled, together with CMU, we focused on determining what new APIs and tools would be needed to export an Architecture Modeling package from our mutual technologies to the DASADA community, suitable for use there in an externalized infrastructure for instrumenting, measuring, and controlling software. Support for reflective models for both logical and physical architectures must be provided. The major impact to our current Acme infrastructure would be to require extensions to languages (Acme, and xAcme) and internal representations (AcmeLib) to support dynamism. We would also need to expand the nature of the constraints that can be specified on architectures as they evolve during the running of a system. Unfortunately, this contribution to a vision for DASADA Phase II was not supported by DARPA.

Key Personnel:

During the Acme project's lifetime key personnel from CMU included: David Garlan, Bob Monroe, and Drew Kompanek. David Wile was the key person from the Software Sciences Division at ISI (later

.

Teknowledge Corp) on the Acme project. Acme supported Bob Monroe's PhD. Dissertation, entitled *"Rapid Development of Custom Software Architecture Design Environments"* August, 1999, from the School of Computer Science, Carnegie Mellon University.

Promotional Activities:

Acme became quite well known during the project's duration in large part because of the extensive proselytizing by its designers, especially David Garlan. A sampling of the trips and talks given includes:

The Second International Software Architecture Workshop held in conjunction with the Foundations of Software Engineering Conference was attended by David Garlan, who discussed preliminary ideas on how to represent systems in different architectural styles, and by David Wile, who presented ideas on Acme semantics.

David Garlan participated as panelist at SEI Symposium, Aug 26, 1997, on a panel entitled: "Architectural Languages and Design Environments: From Research to Practice."

In September, 1997, David Garlan gave talks about Acme-related research at:

- University of Washington (Seattle WA) "Software Architecture: Practice and Potential"
- University of California at Berkeley (Berkeley, CA) "Software Architecture: Practice and Potential"
- Intel (Santa Clara, CA) "An Overview of Architecture Description Languages"
- HP Labs (Palo Alto, CA) "Software Architecture Research at CMU"

David Garlan attended the 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium, and gave a half-day tutorial "Modeling and Analysis of Software Architecture", Zurich, (September, 1997).

He also attended the Workshop on the Foundations of Component-Based Software Engineering and presented the paper "Specifying Dynamism in Software Architectures" by Robert Allen, Remi Douence, and David Garlan (September, 1997).

In July of 1998, David Wile presented an Acme Status Report to MCC's ASSERTT '98 meeting in Chicago, Illinois. Two MCC members joined the Acme users' group as a result.

David Wile presented Acme to the working group meeting of the IFIP WG2.1, programming calculi, on several occasions. The Acme work was discussed extensively, since the group is interested in branching out to design calculi for less well-understood areas of programming (than applicative programs).

David Wile attended a working conference on architectural-based certification of components held in conjunction with ICSE, 2001, in Toronto, where he presented a paper and headed one session. Noteworthy was how important the role of software architecture styles must become in making components certifiably reusable in different contexts.

David Wile attended the Requirements Engineering 01 conference in Toronto, where he presented a paper entitled: "Residual Requirements and Architecture Residues." Several talks at the conference indicated widespread interest in software architecture description during even the early requirements design phase. Most notably, in her keynote presentation Pamela Zave described how she is using an architectural style to reason about telephony feature interactions.

David Wile attended the Working Conference on Complex and Dynamic Systems Architectures," in Brisbane, Australia, where he presented the paper: "Using Dynamic Acme."

Both David Garlan and David Wile attended the numerous EDCS PI meetings, playing major roles in leading committees on various aspects of architecture specification, including especially, formalization of ADLs, tool support for ADLs, language-neutral event formalisms for inclusion in ADLs, and support for reasoning about dynamic architectural aspects of systems.

.

EDCS Community Activities:

Acme was developed in part to provide ADL support for interested EDCS researchers as well as to encourage community consensus among those who were actively pursuing research interests in ADLs. Hence, at the PI meetings David Garlan and David Wile generally took leadership roles in working group meetings. A sampling of such contributions includes:

- Participation in the intra-cluster activity of producing features lists and understanding how we meshed with the other architectural clients in EDCS. We decided that Acme is the nexus of the architectural cluster -- everything relates to it. For example, translators to and from UniCon, Aesop, Wright, Rapide, and GenVoca are all likely possible features for future delivery.
- At ISI, David Wile sat in on part of the Rationale Cluster activities, where we (Acme) were "encouraged" (by John Salasin) to interact with the FLEA group and several others were given pointers to us.
- The EDCS Architecture Cluster semantics subgroup began to design a query-type based framework for assessing the needs of a potential ADL user just trying to decide on an ADL for the problem class at hand. The perceived needs for different analysis types will then be used to determine the most appropriate ADL on which to base the development of the project.
- Paul Hudak (hudak@cs.yale.edu) and David Wile collaborated to study the potential for expressing Acme styles and Acme dynamic semantics in terms of combinators in Haskell. We started out with a formulation of (something like) the Acme Lib primitives in a Haskell monad. A related thread arose in connection with Dick Taylor's (taylor@cs.uci.edu) C2 architecture.
- Richard Brenner of Draper Labs used the Common Lisp Acme Semantics support facilities, working on Draper's version of Common Lisp.

A list of EDCS researchers using Acme at some time during the project's history includes:

- The Acme Tool Developer's Library (AcmeLib) has been downloaded and registered by people at the following organizations. Projects or purposes for which they are using the AcmeLib are listed where they are known.
  - Keysoft Inc. Secure Fault-Tolerant Architectures. Franklin Webber, contact
  - Georgia Tech MORALE project. Bob Waters, contact.
  - Vanderbilt Measurement and Computing Systems Laboratory. Model-Integrated Computing Environments project uses Acme for software modeling. Janos Sztipanovits and Gabor Karsai, contacts.
  - Georgia Tech Software Architecture Analysis. Pascal Schuchhard and Collin Potts, contacts (possibly the same as the MORALE project).
- C2 project at UCI integrated Acme into C2, including the Armani analyzer. Richard Taylor, contact.
- Acme is the mechanism used for integrating the Wright, UniCon, and Aesop environments produced at Carnegie Mellon University. David Garlan and Mary Shaw, contacts.
- Acme Common Lisp / Dynamic Language connections:
  - Richard Brenner, Draper Labs: investigating Common Lisp Acme for reconfiguring missile software (asb@draper.com).
  - John Paterson and Paul Hudak at Yale in a Haskellized version of Acme for scripting studies.

.

- Lockheed-Martin ADAM project. Dick Creps and Paul Kogut, contacts. (Integrators)
- Synquiry Technologies' FAMILIAR project for capturing design rationale. "We would like to use Acme to represent the device information that we use to represent system rationale, and integrate it with other tools in EDCS." Dean Allemang, contact. (Integrators)

Technology Transition

The following represent just a few of the areas where Acme research has had broader impact in research and industry.

Lockheed Martin used Acme as a core technology for packaging and integrating architecture design tools. Efforts are focused on modeling and analyzing proposed architectures for a new implementation of the DoD Ground Transportation Network (GTN). The modeling is being carried out with multiple architecture description languages and tools (including Rapide, Aesop, Wright, and Unicon), using Acme as the interchange language. Email contacts: dick.creps@lmco.com and kogut@paoli.atm.lmco.com.

Additionally, the INSERT project at the CMU Software Engineering Institute (Lehoczky, Feiler, et al) used Acme as the integration mechanism for a set of tools that augment Meta-H with new reliability analysis capabilities. The project used AcmeStudio as the front-end, and developed a set of analyses based on the Armani constraint language.

MCC produced an XML representation of Acme in an internal study, which later matured to be known as ADML (see http://xml.coverpages.org/adml.html).

David Wile presented a talk entitled “The EDCS Architecture Legacy” at the DASADA kickoff meeting in Santa Fe, New Mexico. The talk revolved about Acme’s role as the focus of the existing technology to be used by DASADA researchers requiring architecture support facilities. Subsequent talks discussed Acme as the starting point for community-wide research endeavors, such as a common language for describing architectural events and methods for mapping architecture descriptions into other design languages, such as UML.

David Wile and Bob Balzer taught a two day training course to U.S. Census Bureau and Bureau of Labor Statistics professionals on how to use a census “instrument” design system they created using the PowerPoint Design Editor (under separate funding from NSF). They created a particular architectural style for this purpose and integrated it with two other COTS tools – Microsoft Access and a Web form designer package – to provide analysis and feedback on survey designs to the experts in the field of survey and form designs.

The PowerPoint Design Editor Dynamic Acme style support package was given to David Wells for experimentation in the DASADA program.

Franklin Webber at Key Software (webber@keysoft.com used Acme to express the relationship between a logical Air Traffic Control architecture and its implementation in software and hardware.

Registered Users:

Industrial registered users:

Northrop Grumman, modeling, simulation, and analysis of avionics communication networks. Jeff Lankford and Marty Cohen, contacts.

Hyundai Information Technology. Unknown project. Nam Kyeongsoon, contact.

.

Other registered external users with unknown affiliations and/or projects:

Marcel Weiher (marcel@system.de), "My primary interest in Acme is for the Software Architecture section of my Diplomarbeit, which is titled 'Approaches to Composition and Refinement in Object Oriented Design'. There is also a strong practical interest in going beyond the object connection facilities already present in OpenStep."

Alvaro Medeiros (alvaro@un3.com)

Vanh Sakounsanong (vsakoun@msn.com)

Academic/Research Laboratory registered users:

SEI Architecture group. Acme is used as an intermediate representation for an architecture recovery and reengineering project. Rick Kazman, contact.

SEI. Architectures for an aircraft simulation project, Larry Howard, contact.

SEI. Translating Meta-H descriptions to Acme and Rapide. Mario Barbacci, contact.

Keio University, Japan. Harada Lab. of Computer Science.

Unknown project. Keisuke Kanamaru, contact.

Fukuoka Institute of Technology, Japan, Dept. of Computer Science and Engineering. Unknown project. Jianjun Zhao, contact.

IRISA Project Solidor. Details of project unknown. Erwan Demairy, contact.

Indian Institute of Science, Bangalore India. Project unknown. R.Lakshmi Narayanan, contact.

External users with interest in the Common Lisp Acme:

J Moore and Nancy Eickelmann at MCC (the former for theorem prover applications)

Evaluation

It is worth revisiting the project goals, milestones, and evaluation criteria discussed earlier. From the Technology Transition section above, it is clear that the project succeeded to a large extent in achieving each of the goals set out:

- Provide a low buy-in architecture description language for new ADL users.
  Use of Acme Studio via the web has increased over time.
- Provide network accessible tools for architecture design, analysis (performance prediction, constraint satisfaction, dynamic monitoring, animation, real-time, nondeterministic event), and archiving:
  Significant use of more sophisticated tools, including Armani, by outsiders are success indicators here.
- Provide an interchange medium to enable inter-ADL tool use.
  The growth of xAcme and ADML indicates significant web usage.
- Provide a base level ADL to be extended to new domains.
  Definition and use of domain-specific styles in the PowerPoint Design Editor continues today.
- Provide a focus for architecture community consensus.
  The Architecture Toolkit and the DASADA vision represent the culmination of community consensus.

Selected Acme *References*

The Acme project supported the writing and / or development of all of the following, in whole or in part:

.

Selected web sites:

Acme website: (See Appendix A)

http://www-2.cs.cmu.edu/~acme

Language Overview: (See Appendix B)

http://www-2.cs.cmu.edu/~acme/language_overview.html

Reference Manual:

http://www-.cs.cmu.edu/afs/cs/project/able/www/AcmeWeb/ACME%20StrawManual.html

AcmeLib programmer's manual:

http://www-2.cs.cmu.edu/afs/cs/project/able/www/AcmeWeb/Java%20AcmeLib%20Manual%201.html

EDCS Architecture Legacy presentation:

http://www.schafercorp-ballston.com/dasada/dasada_kickoff_briefs/WileEDCSArchitectureLegacy.ppt

Unpublished

David Garlan and Zhenyu Wang. A Case Study in Software Architecture Interchange, *Submitted for Publication.*

D. Wile. Integrating Syntaxes and their Associated Semantics (Unpublished, 1999: To be submitted to: ACM Symposium on Applied Computing Programming Languages Track. Mar. 2004)

David Garlan, Andrew Kompanek, John Kenney, David Luckham, Bradley Schmerl, and Dave Wile. An Activity Language for the ADL Toolkit. August 2000.

Jianing Hu. Adding Maps to Acme. August 2000.

Publications:

Robert Allen and David Garlan. A Case Study in Architectural Modeling: The AEGIS System, *Proceedings of the Eighth International Workshop on Software Specification and Design* (IWSSD-8), March 1996.

David Wile. ACME Semantics. *In Joint Proceedings of the Second International Software Architecture Workshop (ISAW2) and the International Workshop on Multiple Perspectives in Software Development* (October, 1996, San Francisco, CA) ACM Press.

Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software* January, 1997. pp. 43-52.

Robert Allen and David Garlan. Formal Modeling and Analysis of the HLA RTI, Proceedings of the 1997 Spring Simulation Interoperability Workshop, Orlando, Florida, March 1997.

Robert Allen and David Garlan. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, July 1997.

Bridget Spitznagel and David Garlan. Architecture-Based Performance Analysis. Submitted for publication September 1997.

David Garlan, Robert Monroe, and Dave Wile Acme: An Architecture Description Interchange Language *Proceedings of CASCON 97*, Toronto, Ontario, November 1997, pp. 169-183.

Ralph Melton and David Garlan. Architectural Unification. *Proceedings of CASCON '97*, November 1997.

David Garlan. Higher-Order Connectors. In Proceedings of the Workshop on Compositional Software Architectures, January, 1998.

.

Robert J. Allen, David Garlan, and James Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), November 1998)*.

C. Ramming and D. Wile, eds*. IEEE TSE Special Issue on Domain-Specific Languages*, May/June 1999. pp. 317-333.

Robert T. Monroe. *Rapid Development of Custom Software Architecture Design Environments.* August 1999 Ph.D. Thesis CMU-CS-99-161.pdf

D. Garlan, R. Monroe, and D. Wile. Architectural descriptions of component-based systems. *In Foundations of Component-Based Systems*, Gary Leavens and Murali Sitaraman, ed.s Kluwer, 2000. (See also: http//www.cs.cmu.edu/~acme/)

D. Wile and R. Balzer. *Survey Instrument Creator (SIC) Training Manual*. Teknowledge Corp. 2000.

D. Wile. Residual Requirements and Architectural Residues. *In Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*. Toronto. Aug, 2001. 194-201.

D. Wile. Modeling architecture description languages using AML. *In Automated Software Engineering. 8*: 2001. 63-88.

D. Wile, Supporting the DSL Spectrum. *Journal of Computing and Information Technology*. CIT 9, 2001 (4) 263-287.

D. Wile. Using Dynamic Acme. *In Proceedings of a Working Conference on Complex and Dynamic Systems Architecture*. Brisbane, Australia. Dec. 2001.

D. Wile. Programming Languages *In Encyclopedia of Software Engineering,* 2nd edition, ed. J. Marciniak. John Wiley & Sons. 2002. 1010-1023.

D. Wile. Lessons Learned from Real DSL Experiments. *In Proceedings of the 36th Hawaii International Conference on System Sciences*. Kona. Jan 2003.

D. Wile. Calculating Requirements: an Approach Based on Architecture Style *Proceedings of the Automated Software Engineering Conference*. Montreal. 2003. (Short paper.) To appear.

D. Wile. Toward a Calculus for Abstract Syntax Trees. In Bird, R and Meertens, L. eds. *Proceedings of a Workshop on Algorithmic Languages and Calculi.* Alsace FR. Chapman and Hill. February, 1997. 324-352.

.

APPENDIX A: The ACME Website

The Acme *Architectural Description Language*

(From: http://www-2.cs.cmu.edu/~acme/)

Acme is a simple, generic software architecture description language (ADL) that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools. This site provides an introduction to Acme along with a collection of useful Acme software and technical information.

Overview of the Acme Project

The Acme project began in early 1995 with the goal of providing a common language that could be used to support the interchange of architectural descriptions between a variety of architectural design tools. Although it is still useful as an architectural interchange language, since the project's inception the Acme language and its supporting toolkit have grown into a solid foundation upon which new software architecture design and analysis tools can be built without the need to rebuild standard infrastructure. Currently, the Acme Language and the Acme Tool Developer's Library (AcmeLib) provide a generic, extensible infrastructure for describing, representing, generating, and analyzing software architecture descriptions.
The Acme language and toolkit provide three fundamental capabilities:

- **Architectural interchange**. By providing a generic interchange format for architectural designs, Acme allows architectural tool developers to readily integrate their tools with other complementary tools. Likewise, architects using Acme-compliant tools have a broader array of analysis and design tools available at their disposal than architects locked into a single ADL.

- **Extensible foundation for new architecture design and analysis tools**. Many, if not most, architectural design and analysis tools require a representation for describing, storing, and manipulating architectural designs. Unfortunately, developing good architectural representations is difficult, time consuming, and costly. Acme can mitigate the cost and difficulty of building architectural tools by providing a language and toolkit to use as a foundation for building tools. Acme provides a solid, extensible foundation and infrastructure that allows tool builders to avoid needlessly rebuilding standard tooling infrastructure. Further, Acme's origin as a generic interchange language allows tools developed using Acme as their native architectural representation to be compatible with a broad variety of existing architecture description languages and toolsets with little or no additional developer effort.

- **Architecture Description**. Acme has emerged as a useful architecture description language in its own right. It provides a straightforward set of language constructs for describing architectural structure, architectural types and styles, and annotated properties of the architectural elements. Although not appropriate for all applications, the Acme architecture description language provides a good introduction to architectural modeling, and an easy way to describe relatively simple software architectures.

This site provides an introduction to and overview of Acme along with a repository of Acme specifications, papers and technical literature, Acme examples, Acme tool developer libraries (AcmeLib's), and freely available tools and software contributed by Acme users.

.

Language and Toolkit Release Status

The current release of the language and design tool infrastructure (version 1.x) provides a stable, baselined language and OO-based API for representing software architecture descriptions. Version 1.x supersedes previous Beta releases (beta versions 2.x and 3.x) of the Acme language and tool developer's library (AcmeLib). Please visit our download area for further release information. Pre-1.0 versions of the language and tools are no longer actively supported.

Acme Team

The Acme project has benefited from the contributions and feedback of many members of the software architecture design community. The principle language design and tool development work for Acme has been undertaken by David Garlan, Bob Monroe, and Drew Kompanek at Carnegie Mellon University, and Dave Wile at USC's Information Sciences Institute. This development process has included significant and frequent input and feedback from many members of the DARPA EDCS project.

.

APPENDIX B: An Overview of Acme

(From: http://www-2.cs.cmu.edu/~acme/language_overview.html)

This document serves as an overview of the capabilities of Acme. It covers the basic language features and includes a few small examples. It is based on excerpts from the paper:

> Acme: An Architecture Description Interchange Language, David Garlan, Robert T. Monroe, David Wile, Proceedings of CASCON '97, November 1997.

This paper provides more detailed coverage of background, capabilities and the goals of Acme. In particular, it includes a discussion of the issues that informed the design of the language. For more detailed information on Acme as an ADL, please see this paper.

Language Features

The Acme language provides the following key features:

1. an *architectural ontology* consisting of seven basic architectural design elements;
2. a flexible *annotation mechanism* supporting association of non-structural information using externally defined sublanguages;
3. a *type mechanism* for abstracting common, reusable architectural idioms and styles; and
4. an *open semantic framework* for reasoning about architectural descriptions.

Acme Design Element Types

Acme is built on a core ontology of seven types of entities for architectural representation: *components, connectors, systems, ports, roles, representations, and rep-maps*. These are illustrated in Figures 3 and 4. Of the seven types, the most basic elements of architectural description are *components*, *connectors*, and *systems*.

- *Components* represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include such things as clients, servers, filters, objects, blackboards, and databases.
- *Connectors* represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. Informally they provide the "glue" for architectural designs, and intuitively, they correspond to the lines in box-and-line descriptions. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. But connectors may also represent more complex interactions, such as a client-server protocol or a SQL link between a database and an application.
- *Systems* represent configurations of components and connectors.

Components' interfaces are defined by a set of *ports*. Each port identifies a point of interaction between the component and its environment. A component may provide multiple interfaces by using different types of ports. A port can represent an interface as simple as a single procedure signature, or more complex interfaces, such as a collection of procedure calls that must be invoked in certain specified orders, or an event multi-cast interface point.
Connectors also have interfaces that are defined by a set of *roles*. Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as the *caller* and *callee* roles of an RPC connector, the *reading* and *writing* roles of a pipe, or the *sender* and *receiver* roles of a message passing connector. Other kinds of connectors

.

may have more than two roles. For example an event broadcast connector might have a single *event-announcer* role and an arbitrary number of *event-receiver* roles.

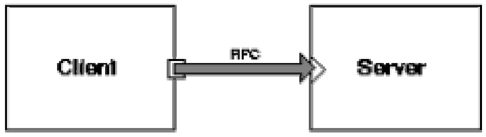


Figure1: Simple Client-Server Diagram

```
System simple_cs = {
Component client = { Port send-request; };
Component server = { Port receive-request; };
Connector rpc = { Roels { caller, callee}};
Attachments {
client.send-request to rpc.caller;
server.receive-request to rpc.callee;
}
```

Figure2: Simple Client-Server System in Acme

As a simple illustrative example, Figure 1 shows a trivial architectural drawing containing a client and server component, connected by an RPC connector. Figure 2 contains its Acme description. The *client* component is declared to have a single *send-request* port, and the server has a single *receive-request* port. The connector has two roles designated *caller* and *callee*. The topology of this system is declared by listing a set of *attachments*.

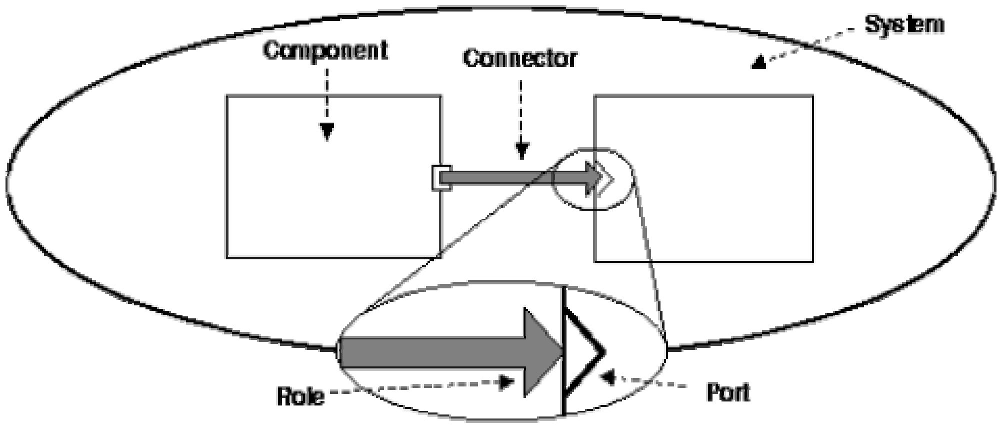


Figure3: Elements of an Acme Description

.

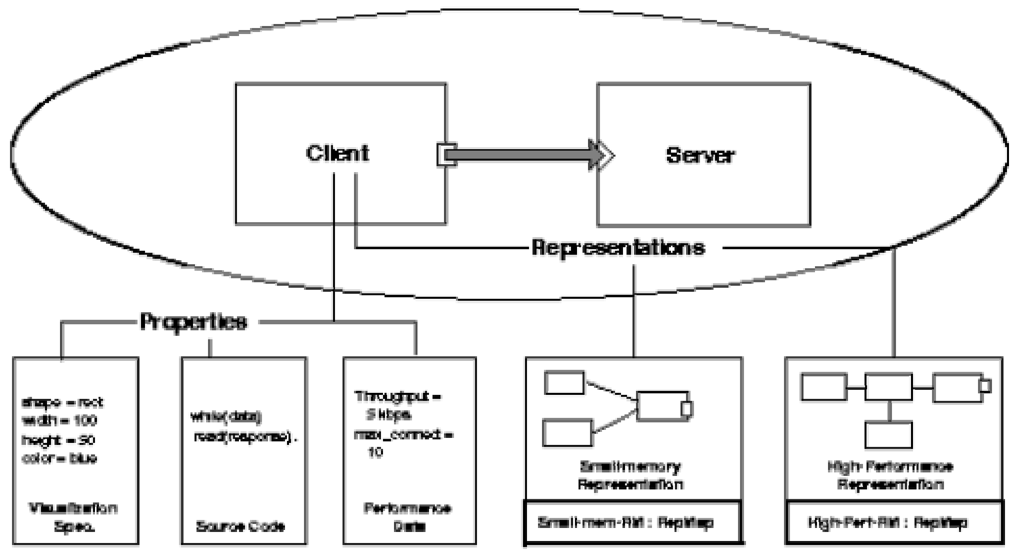


Figure4: Representations and Properties of a Component

Acme supports the hierarchical description of architectures. Specifically, any component or connector can be represented by one or more detailed, lower-level descriptions. (See Figure 4 .) Each such description is termed a *representation* in Acme. The use of multiple representations allows Acme to encode multiple views of architectural entities (although there is nothing built into Acme that supports resolution of inter-view correspondences). It also supports the description of encapsulation boundaries, as well as multiple refinement levels.

When a component or connector has an architectural representation there must be some way to indicate the correspondence between the internal system representation and the external interface of the component or connector that is being represented. A *rep-map* (short for ``representation map'') defines this correspondence. In the simplest case a rep-map provides only an association between internal ports and external ports (or, for connectors, internal roles and external roles). In other cases the map may be considerably more complex. For those cases the rep-map is essentially a tool-interpretable placeholder---similar to the use of properties described in the following section.

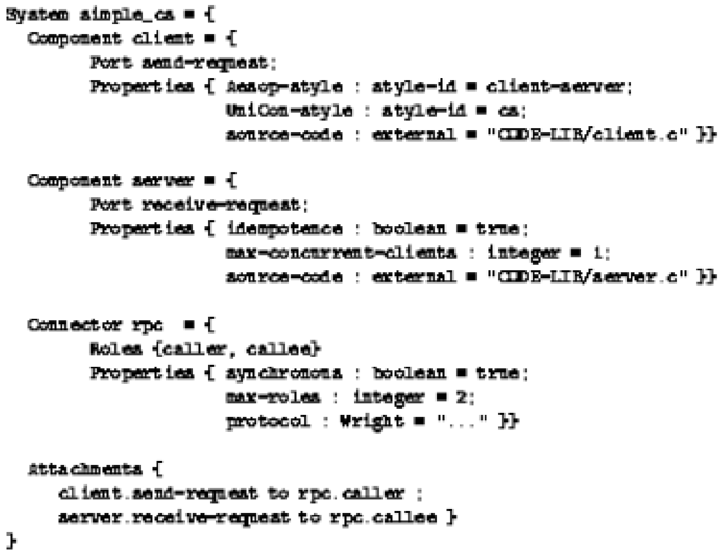
Acme Properties

The seven classes of design element outlined above are sufficient for defining the *structure* of an architecture as a hierarchical graph of components and connectors.

But there is clearly more to architectural description than structure. As discussed earlier, currently there is little consensus about exactly what should be added to the structural information: each ADL typically has its own set of auxiliary information that determines such things as the run-time semantics of the system, detailed typing information (such as types of data communicated between components), protocols of interaction, scheduling constraints, and information about resource consumption.

To accommodate the wide variety of auxiliary information Acme supports annotation of architectural structure with lists of properties. Each property has a name, an optional type, and a value. Any of the seven kinds of Acme architectural design entities can be annotated. Figure 4 shows several properties attached to a hypothetical architecture.

.



```
System simple_cs = {
  Component client = {
        Port send-request;
        Properties { Aesop-style : style-id = client-server;
                     UniCon-style : style-id = cs;
                     source-code : external = "CODE-LIB/client.c" }}

  Component server = {
        Port receive-request;
        Properties { idempotence : boolean = true;
                     max-concurrent-clients : integer = 1;
                     source-code : external = "CODE-LIB/server.c" }}

  Connector rpc  = {
        Roles {caller, callee}
        Properties { synchronous : boolean = true;
                     max-roles : integer = 2;
                     protocol : Wright = "..." }}

  Attachments {
     client.send-request to rpc.caller ;
     server.receive-request to rpc.callee }
}
```

Figure5: Client-Server System with Properties in Acme

From Acme's point of view the properties are uninterpreted values. Properties become useful only when a tool makes use of them for analysis, translation, and manipulation. In Acme the "type" of a property indicates a "sublanguage" with which the property is specified. Acme itself predefines simple types such as integer, string, and boolean. Other types must be interpreted by tools: these tools use the "name" and "type" indicator to figure out whether the value is one that they can process. The default behavior of a tool that does not understand a specific property or property type should be to leave it uninterpreted but preserve it for use by other tools. This is facilitated by requiring standard property delimiter syntax so that a tool can know the extent of a property without having to interpret its contents.

Figure 5 shows the simple client-server system elaborated with several properties. For example, several of the properties indicate how the elements relate to constructs in target ADLs---such as Aesop and UniCon styles. Likewise, the "protocol" property of the RPC connector is declared to be in the "Wright" language and would only be meaningful to a tool that knows how to process that language.

Of course, in order for properties to be useful when interchanged between different ADLs, there must be a common understanding of their meaning. As we have noted, Acme does not explicitly define those meanings, but it does allow for the shared use of properties when those meanings do exist. We anticipate that over time Acme will serve as a vehicle for conventionalization of properties that are useful to more than one ADL.

Several property sublanguages are currently being developed. One is a standard for specifying visualization properties to be used by graphical editors to display architectural descriptions. Another sublanguage is being developed to describe temporal constraints on an architectural description. Details of these sublanguages are beyond the scope of this report.

Acme Families

The Acme features described thus far are sufficient to define an architectural instance, and, in fact, form the basis for the core capabilities of Acme parsing and unparsing tools. As a representation that is good for humans to read and write, however, these features leave much to be desired. Specifically, they provide no facilities for abstracting architectural structure. As a result, common structures in complex system descriptions need to be repeatedly specified. Consider, for example, extending the simple client-server system described in Figure 5 to include multiple clients and multiple servers. Although there is significant common structure underlying

.

each of the clients and servers in the design, the language facilities presented thus far would require the architect to explicitly specify this structure for each design element.
To address this problem the Acme language includes *types*, a mechanism for the specification of recurring component, connector, port and role structures. These types can then be instantiated within the systems in a Acme description. A Family in Acme includes a set of type definitions which define the common structures or design vocabulary system.
NOTE: The Acme Language also includes a template mechanism which is not described here. Please see the original paper for more information on templates.
Although Acme is not intended to be a full-fledged ADL, the addition of families greatly enhances the readability and abstraction capabilities of the language.

Acme's Open Semantic Framework

Acme is primarily concerned with the architectural structure of systems, and hence does not embody specific computational semantics for architectures. Rather, Acme relies on an open semantic framework that provides a basic structural semantics while allowing specific ADLs to associate computational or run-time behavior with architectures using the property construct.
The open semantic framework provides a straightforward mapping of the structural aspects of the language into a logical formalism based on relations and constraints. In this framework, an Acme specification represents a derived predicate, called its prescription. This predicate can be reasoned about using logic or it can be compared for fidelity with real world artifacts that the specification is intended to describe.
To illustrate, consider the simple client-server example architecture specification of Figures 1 and 2 , where a client is linked to a server through a single connector. This system has the following prescription:

```
  exists  client, server, rpc |
    component(client) ^
    component(server) ^
    connector(rpc) ^
    attached(client.send-request,rpc.caller) ^
    attached(server.receive-request,rpc.callee)
```

These predicates can be reasoned about using standard first-order logical machinery. They can also be used as the formal specification of an implementation. (In this case, it requires that one be able to find the artifacts *client* and *server* that purport to be components, a connector artifact *rpc*, and attachments that are specified by the predicate.)
This simple translation scheme is, however, not quite sufficient. Two implicit aspects of the specification also need to be included in the prescription: the first is the closed world assumption which states that all components, connectors, ports and roles have been identified by the existential variables in the specification, all attachments have been specified, and that no more exist; Second, the existential variables must refer to distinct entities. With these additions, the example's prescription reads:

```
exists  client, server, rpc |
  component(client) ^
  component(server) ^
  connector(rpc) ^
  attached(client.send-request,rpc.caller) ^
  attached(server.receive-request,rpc.callee) ^
  client != server ^
  server != rpc ^
  client != rpc ^
  (for all y:component (y) =>
                y = client | y = server) ^
```

.

```
  (for all y:connector(y) => y = rpc) ^
  (for all p,q: attached(p,q) =>
       (p=client.send-request ^ q=rpc.caller)
     | (p=server.receive-request ^ q=rpc.callee))
```

In addition to basic structural information, properties also need to be handled. Property names correspond to predicates that take the entity to which the property applies as an argument and return the value of that property name for the given entity. The values of properties are treated as primitive atoms, without their own semantics. So, for example,

```
 Component client = {
   Port send-request;
   Properties {
      Aesop-style : style-id = client-server;
      UniCon-style : style-id = cs} }
```

adds to the prescription the clauses:

```
   Aesop-style(client) = client-server ^
   Unicon-style(client) = cs
```

Although the value of a property is considered an atomic entity in terms of Acme's structural semantics, tools that manipulate and analyze Acme descriptions can interpret the property values as needed. An example of this approach is the protocol property of the RPC connector specified in the "Wright" sublanguage in example 5 .

```
  Connector rpc  = {
        Roles {caller, callee}
        Property protocol : Wright = "..."; }
```

Tools that don't understand the meaning of the Wright sublanguage can ignore the value of this property, processing it as an uninterpreted string. Tools that do understand the Wright sublanguage can interpret the value of the protocol specification to discover more detailed ADL-specific semantics of the connector.

.